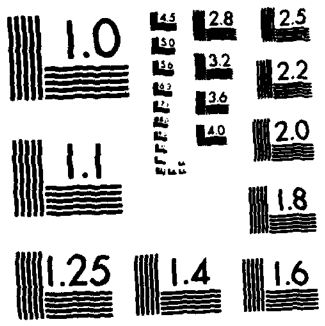
AD-A195 149 A RETROSPECTIVE ON COMMUNICATING SEQUENTIAL PROCESSES 1/1
(U) NEVADA UNIV LAS VEGAS DEPT OF COMPUTER SCIENCE AND
ELECTRICAL.. I A GROSS 18 MAR 88 CSR-88-05
UNCLASSIFIED ARO-24960.11-MA DAAL03-87-G-0004 F/G 12/5 NL

END
DATE
FILMED
8 8-



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ARO 24960.11-MA

(2)

A Retrospective on Communicating Sequential Processes

Todd A Gross*
Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

March 18, 1988

Report CSR-88-05

Department of Computer Science and Electrical Engineering

University of Nevada, Las Vegas
Las Vegas, Nevada 89154

2

A Retrospective on Communicating Sequential Processes

Todd A Gross*
Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas

March 18, 1988

Report CSR-88-05

*Supported by the U. S. Army Research Office under Grant DAAL03-87-G-0004

AD-A195 141

~~UNCLA~~SSIFIED MASTER COPY - FOR REPRODUCTION PURPOSES

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 24960.11-MA
6a. NAME OF PERFORMING ORGANIZATION Univ. of Nevada, Las Vegas; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code) Dept. of Computer Sc. & Elec. Engr. 4505 Maryland Parkway, Las Vegas, NV 89154	7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U. S. Army Research Office; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-87-G-0004
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Retrospective on Communicating Sequential Processes

12. PERSONAL AUTHOR(S) Todd A. Gross

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	March 18, 1988	10

16. SUPPLEMENTARY NOTATION The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Distributed Processing, Computers, Software, Communicating Sequential Processes (CSP), Multiprocess Programs

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In 1978, C. A. R. Hoare published a draft of a programming language called Communicating Sequential Processes (CSP). Ten years later, we can see that this paper had a profound impact on our perception of parallel computation. This paper examines the evolution of CSP over the last 10 years, in order to understand its effect on our current perceptions of parallel and distributed computation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

A Retrospective on Communicating Sequential Processes

Todd Gross

March 9, 1988

1 Introduction

We live in an age of distributed processing. Desktop workstations connected by high-bandwidth networks to one or more central processors and memory stores have become ubiquitous at universities and research-oriented companies. Personal computers connected by local networks are common in smaller companies and colleges. Modern computers can have several, even thousands, of separate processors. As software engineers, we want to be able to utilise the full power of this technology—but at the same time, we want to avoid having to understand the inner complexities. We want to be able to develop programs that run on several processors simultaneously and interdependently, avoiding destructive interactions yet allowing for all necessary and useful constructive interactions.

Since the early 60's, researchers have attempted to provide this power to us, using such techniques as coroutines, critical sections, and semaphores. With the invention of guarded commands [Dij75], we had the ability to create programs that acted *nondeterministically*, yet were *provably* correct. The result of this is that we now had a tool for harnassing the power of parallel processing in a manner that would ensure our processes behaved as specified.

So we could now, using guarded commands and one of several methods of memory protection available (see [PS85]), write parallel programs that worked. But Hoare had the foresight to see that this wouldn't be sufficient for programs running on *separate machines*, because processes running on one machine wouldn't have automatic access to information on other machines. So Hoare devised a protocol for shuttling information between processes. That protocol, in combination with Dijkstra's guarded commands, became CSP.

When Hoare wrote his now famous paper outlining CSP, he had two objectives in mind: to create a spare but powerful method of devising and denoting multiprocess programs, and to allow for the exchange of information between separate machines in a useful and correct manner. In this paper, we will examine CSP as it evolved over the past 10 years and see whether these objectives have been met. We'll also be examining the effect CSP and the research it has spawned have had on software engineering.

2 The Paper that Started it All

In August of 1978, exactly 3 years after Dijkstra's paper on guarded commands was published, Hoare's paper on Communicating Sequential Processes appeared in the Communications of the ACM [Hoa78]. The paper consisted mostly of a BNF grammar for his proposed language and various applications of it. It is worth summarizing each of these parts.

2.1 CSP Grammar

First, we summarize the grammar. A CSP program consists of one or more *processes*, which are independently executed blocks of code. Each process consists of a *label*, a *declaration list* (possibly empty), and a *command list*. In the CSP process below, **P ::** is the label, **integer i;** is the declaration list, and **i := 0;** is the command list:

```
P ::
integer i;
i := 0;
```

Figure 1: A simple CSP process

Labels are straightforward: a name followed by a double colon (::). A name can be subscripted, if one wishes to define several similar processes, like P(1) instead of P.

Declarations are much like Pascal declarations, the only major difference being with *structured* variables. A structured variable consists of an optional label and a list of variables. The label, if given, strongly types the variable, so that

T(x,y) := (a,b)

is not allowed, even if x and a, and y and b, have the same types. A structure consisting of a label and an empty variable list is called a *signal*, and is only used in message passing.

Commands are the equivalent of *statements* in conventional languages. There are 6 basic command types: assignment, parallel, input, output, alternative, and repetitive.

Assignment commands are exactly as in Pascal, but the variables can have a structural type, as in the above example.

Parallel commands consist of a list of processes, separated by double bars (||), and surrounded by brackets. This creates the set of independently executing processes that make up a CSP program. A typical CSP program is just one parallel command.

Input and *output* commands are the protocol Hoare devised to allow processes to send information to each other. The process sending the information issues an output command of the form:

process ! *value*

where *process* is the name of a CSP process and *value* is the value we're sending. The process receiving the information issues an input command of the form:

process ? *variable*

where *variable* is the location the value is stored in. Note that this includes structured variables.

In CSP, the corresponding input and output commands are executed together. Which means that if process A issues an output command to process B, it will not be executed until process B issues the corresponding input command from process A. Likewise, if B issued the input command first, it would have to wait for the corresponding output command from process A. This is the protocol Hoare devised to allow for correct passing of information between distributed processes. There are several ramifications of using this protocol, which will be discussed later.

Alternative and *repetitive* commands are the guarded commands we referred to earlier. They function exactly as discussed in Dijkstra's paper, although Hoare uses a slightly different notation: Where Dijkstra used if...fi to demarcate an alternative command, Hoare uses brackets ([...]), and where Dijkstra uses do...od to demarcate a repetitive command, Hoare uses brackets prepended by an asterisk (*[...]).

2.2 CSP Examples

As one of Hoare's primary goals in writing this paper was to create a simple yet powerful method for designing concurrent programs, he gave several examples used by other researchers in showing the power of their method. For instance, in his paper on coroutines, Conway gave the example of reading in a sequence of cards and sending them to a line printer, substituting carets (^) in the output for double asterisks (**) in the input. Hoare used a straightforward parallel command. He also gave a solution to the dining philosophers problem originally proposed by Dijkstra. In this way, he showed that his method could well be used in situations where other methods of safeguarding memory were shown to be effective.

In addition to these, Hoare gave sample programs that showed that simple processes connected by message passing could work like ordinary subroutines, recursive subroutines (although limited to a predefined level of recursion), and as parallel abstract data types (which means functions can operate on variables of the type concurrently). These suggest that the rather restrictive protocol Hoare devised was powerful enough to perform everything we would require of a concurrent system.

2.3 Implications of the Article

It is clear that Hoare was attempting to create a system that would do for concurrent programs what structured programming and Hoare's Axioms [Hoa69] did for single-stream processing: provide a single structurally sound and logically complete system for generating code to meet desired output specifications. However, we must remember that unlike other proposed methods, no actual CSP system existed. Hoare's proposal was only a draft, a gedankenexperiment. No one knew for sure whether such a system could be built as proposed. Further, there was no proof that even the abstract method worked as expected. So even if such a system were built, we could not be sure it did what we expected it would.

The paper then, in its attempt to provide a simple and elegant method for multiprogramming as well as multiprocessing, left two important questions unanswered: Can we implement it? and Does it do what we think it does? These are the subjects of the next two sections respectively.

3 Can We Implement CSP?

It is perhaps to be expected that a language designed on paper would undergo changes before reaching its final implemented state. Such is indeed the case with CSP: there are several implemented languages that are based on CSP, yet diverge from the original draft. For instance, it should be expected that no one implemented structured variables as Hoare devised them—he himself failed to specify how one declares such variables, they are syntactic sugar and not actually necessary, but most importantly, it is unwise to force a process on an unknown machine to provide a correct abstract type name. We can dismiss this feature as nonessential to the inherent structure of CSP.

It would, however, be fair to say that all present implementations of CSP differ from the original version in one manner that is fundamentally different from the original version. In the original paper, Hoare decided that input and output commands would be *process oriented*. Input commands name the process they receive data from, and output commands name the process they send data to. He foresaw the possibility of making communication *port oriented*, but saw it as "semantically equivalent to the present proposal, provided that each port is connected to exactly one other port in another process" ([Hoa78], page 675). But Silberschats showed that we could look at ports from a broader perspective [Sil81]. To him, a port was a location where any process could "send" or "receive" messages. This is more practical to implement than process oriented messaging, because specifying the process instead of a port forces the compiler to generate the necessary ports. Thus, all three CSP implementations mentioned in [Hul86] (and almost certainly every other implementation) use ports to send and receive messages. As we shall see in the next section, we cannot dismiss this difference so easily.

I mentioned that there have been several implementations of CSP. Instead of discussing all implementations, I will choose the one I am most familiar with, namely occam. It is probably fair to say that occam is the most robust implementation, any feature we see in other implementations we will also see in occam. My information comes from the programming manual [Lim84].

In CSP, a program was essentially a parallel command, which contained the individual processes. Occam does allow one to create named processes, but since communication is port oriented, rather than process oriented, one need not name the processes explicitly. There is a parallel command[1] in occam, like CSP, but process scope is determined by indentation instead of name. For example, the occam command

```
PAR
  VAR y1:
  y1 := 1
  VAR y2:
  y2 := 0
```

creates 2 processes, one sets variable y1 to 1, the other sets y2 to 0. Note that y1 and y2 are *local* variables, and cannot be accessed by the other process—as in CSP. Note that although the following occam command *appears* legal, it tries to share a common variable *y* between two processes, which is not allowed.

```
VAR y:
PAR
  y := 1
  y := 0
```

A major difference between occam and CSP is that occam isn't strongly typed, or even weakly typed. Types of variables are determined by the compiler, and not declared by the programmer. Clearly then, one can't have typed signals, although occam has a generic signal called ANY.

Occam has all 6 commands specified in CSP[2], although input and output commands specify ports (called *channels* in occam) instead of processes. Alter-

[1] Actually, what we call commands in CSP are called processes in occam. For the sake of clarity, I will call them commands.

[2] Actually 7, as sequencing must be explicitly specified in occam with the SEQ command.

native commands are done with the ALT command. Repetitive commands are trickier: one must enclose an ALT command in a WHILE command. WHILE is a deterministic conditional looping command (as in conventional programming languages). There is also a deterministic branching command, the IF command. Subscripting as used in the original paper is fully supported via *replicators*, which allow one to subscript processes, channels, and commands.

It would seem that occam has not only met the CSP description (save for structured variables, which we've already dismissed), it has gone beyond it, as we can use both deterministic and nondeterministic commands, interleaved in any manner we like. We also have greater flexibility in communicating through the use of user-defined channels. So why would we concern ourselves with trying to implement a less powerful language? We'll see why in the next section.

4 Does CSP Do What We Think It Does?

When Dijkstra originally devised structured programming, there were several doubts raised. But perhaps the biggest doubt raised was whether one could do everything under structured programming that one could do previously. After all, he was reducing the set of programs one could write by a sizeable amount. Maybe some programs wouldn't map into the reduced set. Fortunately, Bohm and Jacopini were able to prove that any program we could write previously could be written under structured programming with no loss of functionality [BJ66].

Hoare's paper on CSP has the same effect regarding multiprocess programs: he reduced the set of possible concurrent programs to ones using guarded commands and a message-oriented process-to-process protocol, with no buffering. How do we know that every program we would ever want to write could be done using CSP?

That's a good question. But there's an even better one: how do I know that this CSP program I wrote will do what I think it will? You see, there are problems that can occur in multiprocess programs that have no correlate in single-stream programs like deadlock and starvation. These are problems that cannot be abstracted away by a program canonization, because there are programs that we want to write that have the possibility of deadlocking or starving one of their members. Resource allocation programs for an operating system are but one example. So theoretical analysis of CSP has concentrated on this aspect.

For instance, Levin and Gries wrote a paper in 1981 which gave a proof technique for CSP programs, although they altered the definition slightly [LG81]. The proof is based on Hoare's axiomatic method for single-stream programs [Hoa69], but extended with a proof that the program is free from deadlock. Apt, Francez, and de Roever did a similar proof [AFdR80]. Each program must be proven individually, and as with Hoare's Axioms, it is not yet feasible to devise an automatic theorem prover. But it is even worse for multiprocess programs, for the reasons given above. If a process is added to a program we've already proven correct, we may have to start the proof all over: since the addition of a new process can create deadlocks where none existed before.

Hoare himself later added substantially to the theory of CSP by publishing a book [Hoa85]. At this point, Hoare had the same benefit of hindsight that we do. Every piece of research I have mentioned took place before he wrote the book. He made two major changes to his theory. One was the change from process-oriented to port-oriented communication we discussed earlier, although his ports are unidirectional one-input-one-output channels. The other change was from an assumption of process termination to indefinite execution in parallel commands. At the time, he wanted to be able to use postconditions to prove correctness of the code, as is done with Hoare's Axioms in conventional programs. Later, he was able to find a more satisfactory way of proving correctness, thus he abandoned this rule.

The book is not a *proof* of CSP, it is an *axiomatic system* based on the original BNF grammar. Some things were changed, for instance guarded commands were now represented by a choice operator and the guard was eliminated. The guard, of course, is necessary in the actual program to allow the programmer to determine when a certain action will happen. But if we're only trying to prove the correctness of guarded commands, we can assume a generic guard without loss of generality. A lot has been added: for instance, notations for traces of a program or process, for various types of interactions between processes, and for general nondeterminism. But now, instead of being a set of executable statements that produce a desired result, a CSP program is a set of sequential streams of abstract events. And it is not entirely clear how one converts a CSP program to a set of sequences of abstract events. This is a problem with axiomatic systems in general: making sure that the universe the system works under maps to the universe of interest.

5 Conclusion

CSP, when originally designed, was to serve two purposes —one practical and one prophetic. The practical idea was to create an efficient paradigm for working on multiprocess programs. Hoare's approach had two main focal points: guarded commands and message-based interprocess communication. Both of these have been incorporated in several languages, including one that is used for practical programming (namely occam). But in a practical sense, we can't really say that CSP is a standard. The language Mesa, for instance, is based on *monitors*, which have shared memory. Theoretically, however, CSP *is* a standard. Research on multiprocess systems, particularly distributed multiprocess systems, often use CSP as a model. For instance, Jalote and Campbell used CSP to research fault-tolerance in interactive multiprocess programs [JC86]. This is perhaps to be expected, as CSP was the first proof system to incorporate distributed processes[3].

Which brings us to the prophetic part. In 1978, when Hoare originally devised CSP, there were no distributed programs. One could of course send messages between machines using some sort of mail facility, but each machine ran a separate mailing program. It hadn't occurred to the general computing populace to partition a program and run it on separate *computers*— after all, intracomputer links were faster and more reliable than intercomputer links, and memory could be shared between the processes. But since then, we have found uses for distributed programming. For instance, when one works with an interactive editor on a personal workstation attached by network to a central computer, the workstation will do much of the editing operations itself, and send updates or commands it can't perform to the central computer. While Hoare foresaw the possible uses of distributed processing, he could not have predicted the amount of research that would be done over the next 10 years. CSP has shown itself to be a powerful, flexible, and tractable paradigm; and we are likely to see it used for many years to come.

6 Acknowledgments

The author would like to thank C A R Hoare for reading a preliminary version of the paper and making several useful comments, many of which were incorporated in this paper. The author would also like to thank Tom Nartker, who suggested the topic and provided materials used in the course of research. Finally, the author owes a great deal of thanks to the Army Research Office, who sponsored the publication of this technical report.

[3]Owicki and Gries developed a proof for multiprocess programs in 1976, but they used a **centralised** model.

References

[AFdR80] K Apt, N Francez, and W de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359-385, 1980.

[BJ66] C Bohm and G Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366-371, 1966.

[Dij75] E W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453-457, 1975.

[Hoa69] C A R Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576-580, 1969.

[Hoa78] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666-677, 1978.

[Hoa85] C A R Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Hul86] M E C Hull. Implementation of the CSP notation for concurrent systems. *The Computer Journal*, 29(6):500-505, 1986.

[JC86] P Jalote and R H Campbell. Atomic actions for fault-tolerance using CSP. *IEEE Transactions on Software Engineering*, SE-12(1):59-68, 1986.

[LG81] G M Levin and D Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281-302, 1981.

[Lim84] INMOS Limited. *occam Programming Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[PS85] J L Peterson and A Silberschatz. *Operating System Concepts*, chapter 10. Addison-Wesley, Reading, MA, 2 edition, 1985.

[Sil81] A Silberschatz. Port directed communication. *The Computer Journal*, 24(1):78-82, 1981.

TE

MED

8